Tutorial: Lighting

Tutorial 1. Diffuse Lighting

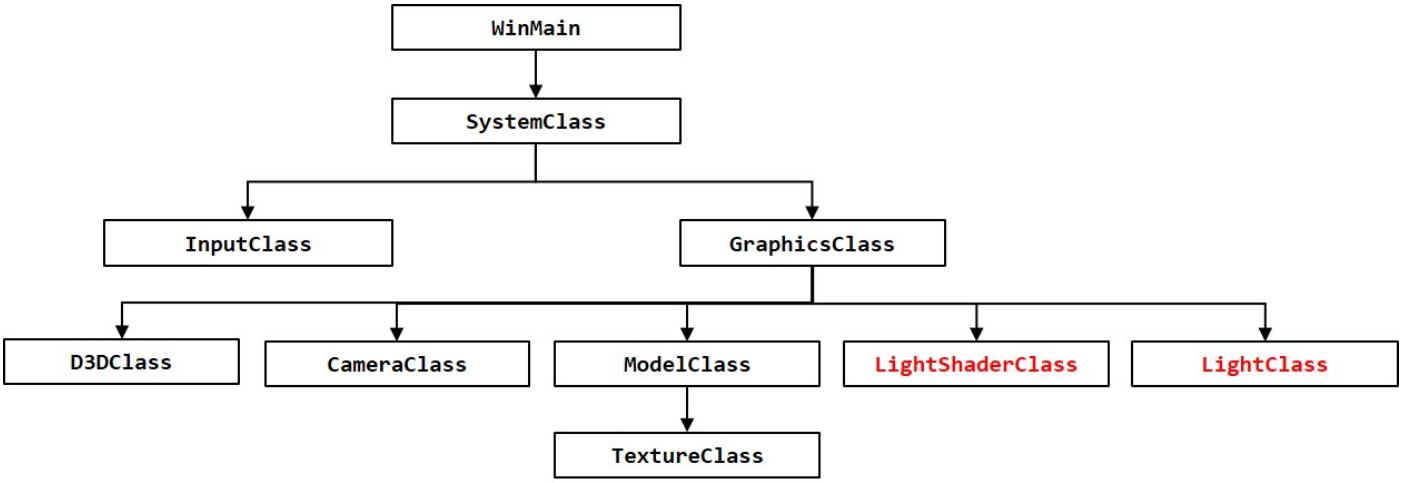
In this tutorial, we will cover how to light 3D objects using **diffuse** lighting and DirectX 11. We will start with the code from the previous tutorial and modify it.

The type of diffuse lighting we will be implementing is called **directional** lighting. Directional lighting is similar to how the Sun illuminates the Earth. It is a light source that is a great distance away and based on the direction it is sending light you can determine the amount of light on any object. However, unlike **ambient** lighting (another lighting model we will cover soon) it will not light up surfaces it does not directly touch.

We picked directional lighting to start with because it is very easy to debug visually. Also since it only requires a direction the formula is simpler than the other types of diffuse lighting such as spot lights and point lights.

The implementation of diffuse lighting in DirectX 11 is done with both vertex and pixel shaders. Diffuse lighting requires just the direction and a normal vector for any polygons that we want to light. The direction is a single vector that you define, and you can calculate the normal for any polygon by using the three vertices that compose the polygon. In this tutorial we will also implement the color of the diffuse light in the lighting equation.

Framework



For this tutorial we will create a new class called LightClass which will represent the light sources in the scenes. LightClass won't do anything other than hold the direction and color of the light. We will also remove the TextureShaderClass and replace it with LightShaderClass which handles the light shading on the model.

We will start the code section by looking at the HLSL light shader. You will notice that the light shader is just an updated version of the texture shader from the previous tutorial (Texture.vs).

light.vs

Both structures now have a 3-float normal vector. The normal vector is used for calculating the amount of light by using the angle between the direction of the normal and the direction of the light.

```
// Type definitions
struct VertexInputType
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};

struct PixelInputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
```

```
};

// Vertex Shader
PixelInputType LightVertexShader(VertexInputType input)
{
```

The normal vector for this vertex is calculated in world space and then normalized before being sent as input into the pixel shader.

```
    // Calculate the normal vector against the world matrix only.
    output.normal = mul(input.normal, (float3x3)worldMatrix);

    // Normalize the normal vector.
    output.normal = normalize(output.normal);

    return output;
}
```

light.ps

We have two new global variables inside the LightBuffer that hold the diffuse color and the direction of the light. These two variables will be set from values in the new LightClass object.

```
cbuffer LightBuffer
{
    float4 diffuseColor;
    float3 lightDirection;
    float padding;
};

struct PixelInputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
};

// Pixel Shader
float4 LightPixelShader(PixelInputType input) : SV_TARGET
{
    float4 textureColor;
    float3 lightDir;
    float lightIntensity;
    float4 color;

    // Sample the pixel color from the texture using the sampler at this texture coordinate location.
    textureColor = shaderTexture.Sample(SampleType, input.tex);
```

This is where the lighting equation that was discussed earlier is now implemented. The light intensity value is calculated as the dot product between the normal vector of triangle and the light direction vector.

```
    // Invert the light direction for calculations.
    lightDir = -lightDirection;

    // Calculate the amount of light on this pixel.
    lightIntensity = saturate(dot(input.normal, lightDir));
```

Finally, the diffuse value of the light is combined with the texture pixel value to produce the color result.

```
    // Determine the final amount of diffuse color based on the diffuse color combined with the light intensity.
    color = saturate(diffuseColor * lightIntensity);

    // Multiply the texture pixel and the final diffuse color to get the final pixel color result.
    color = color * textureColor;
```

```
    return color;
}
```

lightshaderclass.h

The new LightShaderClass is just the TextureShaderClass from the previous tutorials re-written slightly to incorporate lighting.

```
#ifndef _LIGHTSHADERCLASS_H_
#define _LIGHTSHADERCLASS_H_

class LightShaderClass
{
```

The new LightBufferType structure will be used for holding lighting information. This typedef is the same as the new typedef in the pixel shader. Do note that we add an extra float for size padding to ensure the structure is a multiple of 16. Since the structure without an extra float is only 28 bytes CreateBuffer would have failed if we used a sizeof(LightBufferType) because it requires sizes that are a multiple of 16 to succeed.

```
private:
        struct LightBufferType
        {
                D3DXVECTOR4 diffuseColor;
                D3DXVECTOR3 lightDirection;
                float padding;    // Added extra padding so structure is a multiple of 16 for CreateBuffer function requirements.
        };

public:
        LightShaderClass();
        LightShaderClass(const LightShaderClass&);
        ~LightShaderClass();

private:
    bool SetShaderParameters(ID3D11DeviceContext*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX,
            ID3D11ShaderResourceView*, D3DXVECTOR3, D3DXVECTOR4);
```

There is a new private constant buffer for the light information (color and direction). The light buffer will be used by this class to set the global light variables inside the HLSL pixel shader.

```
private:
        ID3D11Buffer* m_lightBuffer;
};

#endif
```

lightshaderclass.cpp

```
#include "lightshaderclass.h"

LightShaderClass::LightShaderClass()
{
```

Set the light constant buffer to null in the class constructor.

```
        m_lightBuffer = 0;
}

LightShaderClass::LightShaderClass(const LightShaderClass& other)
{
}

LightShaderClass::~LightShaderClass()
{
}
```

```
bool LightShaderClass::Initialize(ID3D11Device* device, HWND hwnd)
{
```

The new light.vs and light.ps HLSL shader files are used as input to initialize the light shader.

```
	// Initialize the vertex and pixel shaders.
	result = InitializeShader(device, hwnd, L"./light.vs", L"./light.ps");
}

void LightShaderClass::Shutdown()
{
}
```

The Render function now takes in the light direction and light diffuse color as inputs. These variables are then sent into the SetShaderParameters function and finally set inside the shader itself.

```
bool LightShaderClass::Render(ID3D11DeviceContext* deviceContext, int indexCount, D3DXMATRIX worldMatrix,
	D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView* texture,
	D3DXVECTOR3 lightDirection, D3DXVECTOR4 diffuseColor)
{
	// Set the shader parameters that it will use for rendering.
	result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, texture,
		lightDirection, diffuseColor);
}

bool LightShaderClass::InitializeShader(ID3D11Device* device, HWND hwnd, WCHAR* vsFilename, WCHAR*
	psFilename)
{
```

The polygonLayout variable has been changed to have three elements instead of two. This is so that it can accommodate a normal vector in the layout.

```
	D3D11_INPUT_ELEMENT_DESC polygonLayout[3];
```

We also add a new description variable for the light constant buffer.

```
	D3D11_BUFFER_DESC lightBufferDesc;
```

Load in the new light vertex shader.

```
	// Compile the vertex shader code.
	result = D3DX11CompileFromFile(vsFilename, NULL, NULL, "LightVertexShader", "vs_5_0",
		D3D10_SHADER_ENABLE_STRICTNESS, 0, NULL, &vertexShaderBuffer, &errorMessage,
		NULL);
```

Load in the new light pixel shader.

```
	// Compile the pixel shader code.
	result = D3DX11CompileFromFile(psFilename, NULL, NULL, "LightPixelShader", "ps_5_0",
		D3D10_SHADER_ENABLE_STRICTNESS, 0, NULL, &pixelShaderBuffer, &errorMessage, NULL);
```

One of the major changes to the shader initialization is here in the polygonLayout. We add a third element for the normal vector that will be used for lighting. The semantic name is **NORMAL** and the format is the regular DXGI_FORMAT_R32G32B32_FLOAT which handles 3 floats for the x, y, and z of the normal vector. The layout will now match the expected input to the HLSL vertex shader.

```
	polygonLayout[2].SemanticName = "NORMAL";
	polygonLayout[2].SemanticIndex = 0;
	polygonLayout[2].Format = DXGI_FORMAT_R32G32B32_FLOAT;
	polygonLayout[2].InputSlot = 0;
	polygonLayout[2].AlignedByteOffset = D3D11_APPEND_ALIGNED_ELEMENT;
	polygonLayout[2].InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
	polygonLayout[2].InstanceDataStepRate = 0;
```

Here we setup the light constant **buffer description** which will handle the diffuse light color and light direction. Pay attention to the size of the constant buffers, if they are not multiples of 16 you need to pad extra space on to the end of them or the CreateBuffer function will fail. In this case the constant buffer is 28 bytes with 4 bytes padding to make it 32.

```
	// Setup the description of the light dynamic constant buffer that is in the pixel shader.
	// Note that ByteWidth always needs to be a multiple of 16 if using D3D11_BIND_CONSTANT_BUFFER or
	// CreateBuffer will fail.
	lightBufferDesc.Usage = D3D11_USAGE_DYNAMIC;
	lightBufferDesc.ByteWidth = sizeof(LightBufferType);
	lightBufferDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
	lightBufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
	lightBufferDesc.MiscFlags = 0;
	lightBufferDesc.StructureByteStride = 0;

	// Create the constant buffer pointer so we can access the vertex shader constant buffer in this class.
	result = device->CreateBuffer(&lightBufferDesc, NULL, &m_lightBuffer);
	if (FAILED(result))
	{
		return false;
	}
}

void LightShaderClass::ShutdownShader()
{
```

The new light constant buffer is released in the ShutdownShader function.

```
	// Release the light constant buffer.
	if(m_lightBuffer)
	{
		m_lightBuffer->Release();
		m_lightBuffer = 0;
	}
}

void LightShaderClass::OutputShaderErrorMessage(ID3D10Blob* errorMessage, HWND hwnd, WCHAR*
	shaderFilename)
{
}
```

The SetShaderParameters function now takes in lightDirection and diffuseColor as inputs.

```
bool LightShaderClass::SetShaderParameters(ID3D11DeviceContext* deviceContext, D3DXMATRIX worldMatrix,
	D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView* texture,
	D3DXVECTOR3 lightDirection, D3DXVECTOR4 diffuseColor)
{
```

The light constant buffer is setup the same way as the matrix constant buffer. We first lock the buffer and get a pointer to it. After that we set the diffuse color and light direction using that pointer. Once the data is set, we unlock the buffer and then set it in the pixel shader. Note that we use the PSSetConstantBuffers function instead of VSSetConstantBuffers since this is a pixel shader buffer we are setting.

```
	// Lock the light constant buffer so it can be written to.
	result = deviceContext->Map(m_lightBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedResource);
	if (FAILED(result))
	{
		return false;
	}

	// Get a pointer to the data in the constant buffer.
	dataPtr2 = (LightBufferType*)mappedResource.pData;

	// Copy the lighting variables into the constant buffer.
	dataPtr2->diffuseColor = diffuseColor;
	dataPtr2->lightDirection = lightDirection;
```

```
	dataPtr2->padding = 0.0f;

	// Unlock the constant buffer.
	deviceContext->Unmap(m_lightBuffer, 0);

	// Set the position of the light constant buffer in the pixel shader.
	bufferNumber = 0;

	// Finally set the light constant buffer in the pixel shader with the updated values.
	deviceContext->PSSetConstantBuffers(bufferNumber, 1, &m_lightBuffer);

	return true;
}

void LightShaderClass::RenderShader(ID3D11DeviceContext* deviceContext, int indexCount)
{
}
```

lightclass.h

Now we will look at the new light class which is very simple. Its purpose is only to maintain the direction and color of lights.

```
#ifndef _LIGHTCLASS_H_
#define _LIGHTCLASS_H_

#include <d3dx10math.h>

class LightClass
{
public:
	LightClass();
	LightClass(const LightClass&);
	~LightClass();

	void SetDiffuseColor(float, float, float, float);
	void SetDirection(float, float, float);

	D3DXVECTOR4 GetDiffuseColor();
	D3DXVECTOR3 GetDirection();

private:
	D3DXVECTOR4 m_diffuseColor;
	D3DXVECTOR3 m_direction;
};

#endif
```

lightclass.cpp

```
#include "lightclass.h"

LightClass::LightClass()
{
}

LightClass::LightClass(const LightClass& other)
{
}

LightClass::~LightClass()
{
}

void LightClass::SetDiffuseColor(float red, float green, float blue, float alpha)
```

```
{
	m_diffuseColor = D3DXVECTOR4(red, green, blue, alpha);
	return;
}

void LightClass::SetDirection(float x, float y, float z)
{
	m_direction = D3DXVECTOR3(x, y, z);
	return;
}

D3DXVECTOR4 LightClass::GetDiffuseColor()
{
	return m_diffuseColor;
}

D3DXVECTOR3 LightClass::GetDirection()
{
	return m_direction;
}
```

graphicsclass.h

The GraphicsClass now has two new includes for the LightShaderClass and the LightClass.

```
#include "lightshaderclass.h"
#include "lightclass.h"

class GraphicsClass
{
```

There are two new private variables for the light shader and the light object.

```
private:
	LightShaderClass* m_LightShader;
	LightClass* m_Light;
};
```

graphicsclass.cpp

```
GraphicsClass::GraphicsClass()
{
```

The light shader and light object are set to null in the class constructor.

```
	m_LightShader = 0;
	m_Light = 0;
}

bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)
{
```

The new light shader object is created and initialized here.

```
	// Create the light shader object.
	m_LightShader = new LightShaderClass;
	if(!m_LightShader)
	{
		return false;
	}

	// Initialize the light shader object.
	result = m_LightShader->Initialize(m_D3D->GetDevice(), hwnd);
	if(!result)
```

```
	{
		MessageBox(hwnd, L"Could not initialize the light shader object.", L"Error", MB_OK);
		return false;
	}
```

The new light object is created here.

```
	// Create the light object.
	m_Light = new LightClass;
	if(!m_Light)
	{
		return false;
	}
```

The color of the light is set to purple and the light direction is set to point down the positive Z axis.

```
	// Initialize the light object.
	m_Light->SetDiffuseColor(1.0f, 1.0f, 1.0f, 1.0f);
	m_Light->SetDirection(0.0f, 0.0f, 1.0f);
}

void GraphicsClass::Shutdown()
{
```

The Shutdown function releases the new light and light shader objects.

```
	// Release the light object.
	if(m_Light)
	{
		delete m_Light;
		m_Light = 0;
	}

	// Release the light shader object.
	if(m_LightShader)
	{
		m_LightShader->Shutdown();
		delete m_LightShader;
		m_LightShader = 0;
	}
}

bool GraphicsClass::Render(float rotation)
{
```

The light shader is called here to render the triangle. The new light object is used to send the diffuse light color and light direction into the Render function so that the shader has access to those values.

```
	// Render the model using the light shader.
	result = m_LightShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(), worldMatrix,
		viewMatrix, projectionMatrix, m_Model->GetTexture(), m_Light->GetDirection(),
		m_Light->GetDiffuseColor());
}
```

Summary

With a few changes to the code we were able to implement some basic directional lighting. Make sure you understand how normal vectors work and why they are important to calculating lighting on polygon faces. Note that the back of the spinning triangle will not light up since we have back face culling enabled in our D3DClass.

Exercises

1. Modify the codes in the pixel shader so that the shaderTexture is no longer used and you should see the lighting effect without the texture.
2. Change the color of the light to a different color.
3. Change the direction of the light to go down the positive and the negative X axis and the speed of the rotation faster.

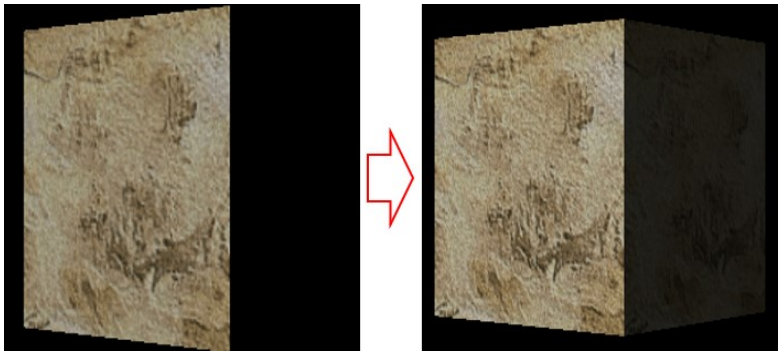
Tutorial 2. Ambient Lighting

This tutorial will be an introduction to using ambient lighting in DirectX 11 with HLSL.

For example, imagine you are in a room and the only light source is sunlight which is coming in from a window. The sunlight does not directly point at all the surfaces in the room, but everything in the room is illuminated to a certain extent due to bouncing light particles. This lighting effect on the surfaces that the sun is not directly pointing at is called **ambient** lighting.

A simple equation is used to simulate ambient lighting. Each pixel is set to the value of the ambient light at the start of the pixel shader. All other operations just add their value to the ambient color. This way, we ensure everything is at a minimum using the ambient color value.

Ambient lighting adds far more realism to a 3D scene. For example, the following picture (the left side) that has only diffuse lighting pointing toward the positive *x*-axis at a cube, does not look realistic due to the lack of lighting on the right side of the cube. The right side of the picture shows the image result if 15% of ambient white light is added to the same scene.



This now gives us a more realistic lighting effect that we as humans are used to.

We will now look at the changes to the code to implement ambient lighting. This tutorial is built on the previous tutorials that used diffuse lighting. We will now add the ambient component with just a few changes.

light.vs

There is no change in the light vertex shader.

light.ps

The light constant buffer is updated with a new 4 float ambient color value. This will allow the ambient color to be set in this shader by outside classes.

```
cbuffer LightBuffer
{
    float4 ambientColor;
    float4 diffuseColor;
    float3 lightDirection;
    float padding;
};

float4 LightPixelShader(PixelInputType input) : SV_TARGET
{
```

We set the output color value to the base ambient color. All pixels will now be illuminated by a minimum of the ambient color value.

```
    // Sample the pixel color from the texture using the sampler at this texture coordinate location.
    textureColor = shaderTexture.Sample(SampleType, input.tex);

    // Set the default output color to the ambient light value for all pixels.
    color = ambientColor;
```

```
    // Invert the light direction for calculations.
    lightDir = -lightDirection;

    // Calculate the amount of light on this pixel.
    lightIntensity = saturate(dot(input.normal, lightDir));
```

Check if the N dot L is greater than zero. If it is then adding the diffuse color to the ambient color. If not then you need to be careful to not add the diffuse color. The reason being is that the diffuse color could be negative and it will subtract away some of the ambient color in the addition which is not correct.

```
    // Determine the final amount of diffuse color based on the diffuse color combined with the light intensity.
    color = saturate(diffuseColor * lightIntensity);

    If (lightIntensity > 0.0f)
    {
        // Determine the final diffuse color based on the diffuse color and the amount of light intensity.
        color += (diffuseColor * lightIntensity);
    }
```

Make sure to saturate the final output light color since the combination of ambient and diffuse could have been greater than 1.

```
    // Saturate the final light color.
    color = saturate(color);

    // Multiply the texture pixel and the final diffuse color to get the final pixel color result.
    color = color * textureColor;

    return color;
}
```

lightshaderclass.h

```
class LightShaderClass
{
```

The LightBufferType has been updated to have an ambient color component.

```
private:
        struct LightBufferType
        {
                D3DXVECTOR4 ambientColor;
                D3DXVECTOR4 diffuseColor;
                D3DXVECTOR3 lightDirection;
                float padding;
        };
};
```

lightshaderclass.cpp

The Render function now takes in an ambient color value which is then sets in the shader before rendering.

```
bool LightShaderClass::Render(ID3D11DeviceContext* deviceContext, int indexCount, D3DXMATRIX worldMatrix,
        D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView* texture,
        D3DXVECTOR3 lightDirection, D3DXVECTOR4 ambientColor, D3DXVECTOR4 diffuseColor)
{
        // Set the shader parameters that it will use for rendering.
        result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, texture,
                lightDirection, ambientColor, diffuseColor);
}
```

The SetShaderParameters function now takes in an ambient light color value.

```
bool LightShaderClass::SetShaderParameters(ID3D11DeviceContext* deviceContext, D3DXMATRIX worldMatrix,
        D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView* texture,
        D3DXVECTOR3 lightDirection, D3DXVECTOR4 ambientColor, D3DXVECTOR4 diffuseColor)
{
```

The ambient light color is mapped into the light buffer and then set as a constant in the pixel shader before rendering.

```
        // Copy the lighting variables into the constant buffer.
        dataPtr2->ambientColor = ambientColor;
}
```

lightclass.h

The LightClass was updated for this tutorial to have an ambient component and related helper functions.

```
class LightClass
{
public:
        void SetAmbientColor(float, float, float, float);
        D3DXVECTOR4 GetAmbientColor();

private:
        D3DXVECTOR4 m_ambientColor;
};
```

lightclass.cpp

```
void LightClass::SetAmbientColor(float red, float green, float blue, float alpha)
{
        m_ambientColor = D3DXVECTOR4(red, green, blue, alpha);
        return;
}

D3DXVECTOR4 LightClass::GetAmbientColor()
{
        return m_ambientColor;
}
```

graphicsclass.h

The header for the GraphicsClass hasn't changed for this tutorial.

graphicsclass.cpp

```
bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)
{
```

Set the intensity of the ambient light to 15% white color. Also set the direction of the light to point down the positive X axis so we can directly see the effect of ambient lighting on the cube.

```
        // Initialize the light object.
        m_Light->SetAmbientColor(0.15f, 0.15f, 0.15f, 1.0f);
        m_Light->SetDirection(1.0f, 0.0f, 0.0f);

        return true;
}

bool GraphicsClass::Frame()
{
```

We have slowed down the rotation by half so the effect is easier to see.

```
        // Update the rotation variable each frame.
```

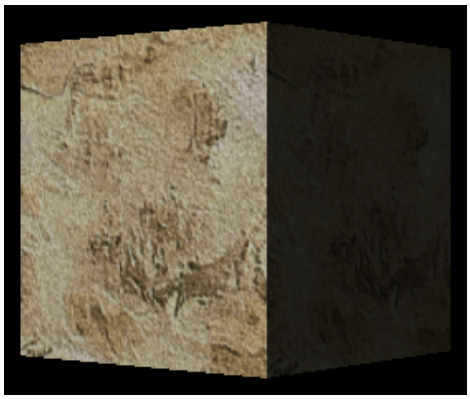
```
        rotation += (float)D3DX_PI * 0.005f;
}

bool GraphicsClass::Render(float rotation)
{
```

The light shader now takes in as input the ambient color of the light.

```
        // Render the model using the light shader.
        result = m_LightShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(), worldMatrix,
                viewMatrix, projectionMatrix, m_Model->GetTexture(), m_Light->GetDirection(),
                m_Light->GetAmbientColor(), m_Light->GetDiffuseColor());
}
```

Summary



With the addition of ambient lighting all surfaces now illuminate to a minimum degree to produce a more realistic lighting effect.

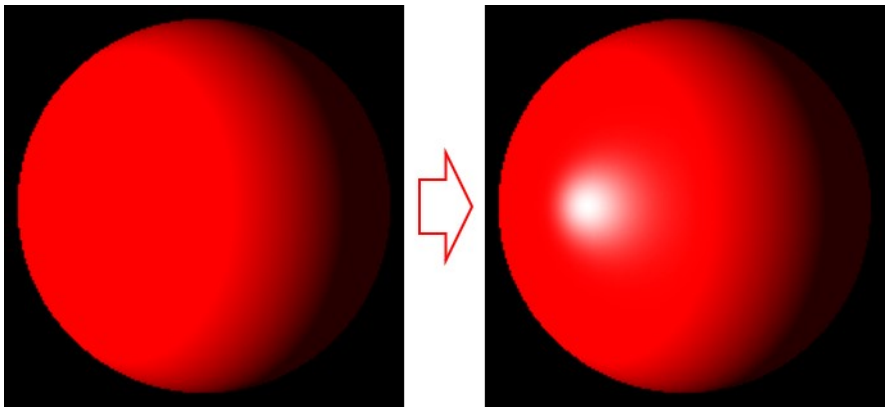
Exercises

1. Change the ambient light value to see just the diffuse component.
2. Comment out the color = color * textureColor; line in the pixel shader to see just the lighting effect.

Tutorial 3. Specular Lighting

This tutorial will be an introduction to using specular lighting in DirectX 11 with HLSL. The code in this tutorial builds on the code from the previous tutorial.

Specular lighting is the use of bright spot highlights to give visual clues for light source locations. For example, a red sphere with just ambient and diffuse lighting looks like the following (left), and if we add a white specular highlight, we get the following result:



Specular lighting is most commonly used to give light reflection from metallic surfaces such as mirrors and highly polished/reflective metal surfaces. It is also used on other materials such as reflecting sunlight off of water. Used well it can add a degree of photo realism to most 3D scenes.

The equation for specular lighting is the following:

$$\textbf{Specular Lighting} = C_s * sum[L_S * (N \cdot H)^P * \textbf{Attent} * \textbf{Spot}]$$

Parameter	Default value	Type	Description
C_s	(0,0,0,0)	D3DCOLORVALUE	Specular color
sum	N/A	N/A	Summation of each light's specular component
N	N/A	D3DVECTOR	Vertex normal
H	N/A	D3DVECTOR	Half way vector: a unit vector exactly halfway between the view and the light direction
P	0.0	FLOAT	Specular reflection power. Range is 0 to +infinity
L_s	(0,0,0,0)	D3DCOLORVALUE	Light specular color
Atten	N/A	FLOAT	Light attenuation value. See Attenuation and Spotlight Factor (Direct3D 9)
Spot	N/A	FLOAT	Spotlight factor. See Attenuation and Spotlight Factor (Direct3D 9)

We will modify the equation to produce just the basic specular lighting effect as follows:

$$\textbf{Specular Lighting} = L_S * (V \cdot R)^P$$

Parameter	Default value	Type	Description
V	N/A	D3DVECTOR	Viewing direction
R	N/A	D3DVECTOR	Reflection vector

The reflection vector in this equation has to be produced by multiplying double the light intensity by the vertex normal. The direction of the light is subtracted which then gives the reflection vector between the light source and the viewing angle:

$$R = 2 * L_I * N - L$$

Parameter	Default value	Type	Description
L_I	N/A	FLOAT	Light intensity value
L	N/A	D3DVECTOR	Light direction

The viewing direction in the equation is produced by subtracting the location of the camera by the position of the vertex:

$$\boldsymbol{V = C - P}$$

Parameter	Default value	Type	Description
C	N/A	D3DVECTOR	Camera position
P	N/A	D3DVECTOR	Vertex position

Let's look at the modified light shader to see how this is implemented:

light.vs

We add a new constant buffer to hold camera information. In this shader we require the position of the camera to determine where this vertex is being viewed from for specular calculations.

```
cbuffer CameraBuffer
{
    float3 cameraPosition;
    float padding;
};
```

The PixelInputType structure is modified as the viewing direction needs to be calculated in the vertex shader and then sent into the pixel shader for specular lighting calculations.

```
struct PixelInputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
    float3 viewDirection : TEXCOORD1;
};

PixelInputType LightVertexShader(VertexInputType input)
{
    float4 worldPosition;
```

The viewing direction is calculated here in the vertex shader. We calculate the world position of the vertex and subtract that from the camera position to determine where we are viewing the scene from. The final value is normalized and sent into the pixel shader.

```
    // Normalize the normal vector.
    output.normal = normalize(output.normal);

    // Calculate the position of the vertex in the world.
    worldPosition = mul(input.position, worldMatrix);

    // Determine the viewing direction based on the position of the camera and the position of the vertex in the world.
    output.viewDirection = cameraPosition.xyz - worldPosition.xyz;

    // Normalize the viewing direction vector.
    output.viewDirection = normalize(output.viewDirection);

    return output;
}
```

light.ps

The light buffer has been updated to hold specularColor and specularPower values for specular lighting calculations.

```
cbuffer LightBuffer
{
    float4 ambientColor;
    float4 diffuseColor;
```

```
    float3 lightDirection;
    float specularPower;
    float4 specularColor;
};
```

The PixelInputType structure is modified here as well to reflect the changes to it in the vertex shader.

```
struct PixelInputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
    float3 viewDirection : TEXCOORD1;
};

float4 LightPixelShader(PixelInputType input) : SV_TARGET
{
    float4 textureColor;
    float3 lightDir;
    float lightIntensity;
    float4 color;
    float3 reflection;
    float4 specular;

    // Sample the pixel color from the texture using the sampler at this texture coordinate location.
    textureColor = shaderTexture.Sample(SampleType, input.tex);

    // Set the default output color to the ambient light value for all pixels.
    color = ambientColor;

    // Initialize the specular color.
    specular = float4(0.0f, 0.0f, 0.0f, 0.0f);

    // Invert the light direction for calculations.
    lightDir = -lightDirection;

    // Calculate the amount of light on this pixel.
    lightIntensity = saturate(dot(input.normal, lightDir));

    if(lightIntensity > 0.0f)
    {
        // Determine the final diffuse color based on the diffuse color and the amount of light intensity.
        color += (diffuseColor * lightIntensity);

        // Saturate the ambient and diffuse color.
        color = saturate(color);
```

The reflection vector for specular lighting is calculated here in the pixel shader provided the light intensity is greater than zero. This is the same equation as listed at the beginning of the tutorial.

```
        // Calculate the reflection vector based on the light intensity, normal vector, and light direction.
        reflection = normalize(2 * lightIntensity * input.normal - lightDir);
```

The amount of specular light is then calculated using the reflection vector and the viewing direction. The smaller the angle between the viewer and the light source the greater the specular light reflection will be. The result is taken to the power of the specularPower value. The lower the specularPower value the greater the final effect is.

```
        // Determine the amount of specular light based on the reflection vector, viewing direction, and specular power.
        specular = pow(saturate(dot(reflection, input.viewDirection)), specularPower);
    }

    // Multiply the texture pixel and the input color to get the textured result.
    color = color * textureColor;
```

We don't add the specular effect until the end. It is a highlight and needs to be added to the final value or it will not show up properly.

```
    // Add the specular component last to the output color.
    color = saturate(color + specular);

    return color;
}
```

lightshaderclass.h

The LightShaderClass has been modified from the previous tutorial to handle specular lighting now.

```
class LightShaderClass
{
```

We add a new camera buffer structure to match the new camera constant buffer in the vertex shader. Note we add a padding to make the structure size a multiple of 16 to prevent CreateBuffer failing when using sizeof with this structure.

```
private:
        struct CameraBufferType
        {
                D3DXVECTOR3 cameraPosition;
                float padding;
        };
```

The LightBufferType has been modified to hold a specular color and specular power to match the light constant buffer in the pixel shader. Pay attention to the fact that we placed the specular power by the light direction to form a 4 float slot instead of using padding so that the structure could be kept in multiples of 16 bytes. Also had specular power been placed last in the structure and no padding used beneath light direction then the shader would not have functioned correctly. This is because even though the structure was a multiple of 16 the individual slots themselves were not aligned logically to 16 bytes each.

```
        struct LightBufferType
        {
                D3DXVECTOR4 ambientColor;
                D3DXVECTOR4 diffuseColor;
                D3DXVECTOR3 lightDirection;
                float specularPower;
                D3DXVECTOR4 specularColor;
        };
```

We add a new camera constant buffer here which will be used for setting the camera position in the vertex shader.

```
private:
        ID3D11Buffer* m_cameraBuffer;
};
```

lightshaderclass.cpp

```
LightShaderClass::LightShaderClass()
{
```

Initialize the new camera constant buffer to null in the class constructor.

```
        m_cameraBuffer = 0;
}
```

The Render function now takes in cameraPosition, specularColor, and specularPower values and sends them into the SetShaderParameters function to make them active in the light shader before rendering occurs.

```
bool LightShaderClass::Render(ID3D11DeviceContext* deviceContext, int indexCount, D3DXMATRIX worldMatrix,
        D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView* texture,
```

```
		D3DXVECTOR3 lightDirection, D3DXVECTOR4 ambientColor, D3DXVECTOR4 diffuseColor,
		D3DXVECTOR3 cameraPosition, D3DXVECTOR4 specularColor, float specularPower)
{
	// Set the shader parameters that it will use for rendering.
	result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, texture,
			lightDirection, ambientColor, diffuseColor, cameraPosition, specularColor, specularPower);
}

bool LightShaderClass::InitializeShader(ID3D11Device* device, HWND hwnd, WCHAR* vsFilename, WCHAR*
		psFilename)
{
		D3D11_BUFFER_DESC cameraBufferDesc;
```

We setup the description of the new camera buffer and then create a buffer using that description. This will allow us to interface with and set the camera position in the vertex shader.

```
		// Setup the description of the camera dynamic constant buffer that is in the vertex shader.
		cameraBufferDesc.Usage = D3D11_USAGE_DYNAMIC;
		cameraBufferDesc.ByteWidth = sizeof(CameraBufferType);
		cameraBufferDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
		cameraBufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
		cameraBufferDesc.MiscFlags = 0;
		cameraBufferDesc.StructureByteStride = 0;

		// Create the camera constant buffer pointer so we can access the vertex shader constant buffer from within this class.
		result = device->CreateBuffer(&cameraBufferDesc, NULL, &m_cameraBuffer);
		if(FAILED(result))
		{
			return false;
		}
}

void LightShaderClass::ShutdownShader()
{
```

Release the new camera constant buffer in the ShutdownShader function.

```
		// Release the camera constant buffer.
		if(m_cameraBuffer)
		{
			m_cameraBuffer->Release();
			m_cameraBuffer = 0;
		}
}
```

The SetShaderParameters function has been modified to take as input cameraPosition, specularColor, and specularPower.

```
bool LightShaderClass::SetShaderParameters(ID3D11DeviceContext* deviceContext, D3DXMATRIX worldMatrix,
		D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView* texture,
		D3DXVECTOR3 lightDirection, D3DXVECTOR4 ambientColor, D3DXVECTOR4 diffuseColor,
		D3DXVECTOR3 cameraPosition, D3DXVECTOR4 specularColor, float specularPower)
{
		CameraBufferType* dataPtr3;
```

Here we lock the camera buffer and set the camera position value in it.

```
		// Lock the camera constant buffer so it can be written to.
		result = deviceContext->Map(m_cameraBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0, &mappedResource);
		if(FAILED(result))
		{
			return false;
		}

		// Get a pointer to the data in the constant buffer.
```

```
	dataPtr3 = (CameraBufferType*)mappedResource.pData;

	// Copy the camera position into the constant buffer.
	dataPtr3->cameraPosition = cameraPosition;
	dataPtr3->padding = 0.0f;

	// Unlock the camera constant buffer.
	deviceContext->Unmap(m_cameraBuffer, 0);
```

Note that we set the bufferNumber to 1 instead of 0 before setting the constant buffer. This is because it is the second buffer in the vertex shader (the first being the matrix buffer).

```
	// Set the position of the camera constant buffer in the vertex shader.
	bufferNumber = 1;

	// Now set the camera constant buffer in the vertex shader with the updated values.
	deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m_cameraBuffer);
```

The light constant buffer now sets the specular color and specular power so that the pixel shader can perform specular lighting calculations.

```
	// Copy the lighting variables into the light constant buffer.
	dataPtr2->specularColor = specularColor;
	dataPtr2->specularPower = specularPower;
}
```

lightclass.h

The LightClass has been modified for this tutorial to include specular components and specular related helper functions.

```
class LightClass
{
public:
	void SetSpecularColor(float, float, float, float);
	void SetSpecularPower(float);

	D3DXVECTOR4 GetSpecularColor();
	float GetSpecularPower();

private:
	D3DXVECTOR4 m_specularColor;
	float m_specularPower;
};
```

lightclass.cpp

```
void LightClass::SetSpecularColor(float red, float green, float blue, float alpha)
{
	m_specularColor = D3DXVECTOR4(red, green, blue, alpha);
	return;
}

void LightClass::SetSpecularPower(float power)
{
	m_specularPower = power;
	return;
}

D3DXVECTOR4 LightClass::GetSpecularColor()
{
	return m_specularColor;
}
```

```
float LightClass::GetSpecularPower()
{
	return m_specularPower;
}
```

graphicsclass.h

The header file for the GraphicsClass has not changed for this tutorial.

graphicsclass.cpp

```
bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)
{
```

Object we now set the specular color and the specular power. For this tutorial we set the specular color to white and set the specular power to 32. Remember that the lower the specular power value the greater the specular effect will be.

```
	// Initialize the light object.
	m_Light->SetDirection(0.0f, 0.0f, 1.0f);
	m_Light->SetSpecularColor(1.0f, 1.0f, 1.0f, 1.0f);
	m_Light->SetSpecularPower(32.0f);
}

bool GraphicsClass::Render(float rotation)
{
```

The light shader render function now takes in the camera position, the light specular color, and the light specular power.

```
	// Render the model using the light shader.
	result = m_LightShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(), worldMatrix,
		viewMatrix, projectionMatrix, m_Model->GetTexture(), m_Light->GetDirection(),
		m_Light->GetAmbientColor(), m_Light->GetDiffuseColor(), m_Camera->GetPosition(),
		m_Light->GetSpecularColor(), m_Light->GetSpecularPower());
}
```

Summary

With the addition of specular lighting we now get a bright white flash each time the cube surface evenly faces the camera viewing direction.

Exercises

1. Change the direction of the light such as m_Light->SetDirection(1.0f, 0.0f, 1.0f) to see the effect if the light source is from a different direction.
2. Create a 5000+ poly sphere model with a red texture to recreate the sphere images at the top of the tutorial.

Tutorial 4: Multiple Point Lights

This tutorial will cover how to implement multiple point lights in DirectX 11 using HLSL and C++.

Most of the tutorials we have used directional light since it simpler to understand and debug visually. However, point lights are important for simulating many light sources. Pretty much anything that is illuminated with a light bulb is a point light. Point lights have a position of origin and a color, they do not require a direction. They illuminate strongest at the center or origin and dissipate in an even circular fashion as the light moves away from the source. Point lights also use different types of attenuation and range to simulate how different point lights dissipate more accurately over a distance. We will just cover the basic point light without attenuation and range for this tutorial.

The second concept we will be covering in this tutorial is multiple light sources. Point lights work perfectly with this concept as most scenes only have a single directional light but will have numerous point lights. For example, think about being inside a factory. It may have a single directional light coming through the windows but there will be hundreds of light fixtures used to properly illuminate the building. Also note that DirectX 11 with HLSL has no limitation on the number of point lights you can have in a scene, it is completely up to you.

Light.vs

HLSL allows the use of defines. For this point light tutorial we will define how many point lights this shader can use. Both the vertex shader and pixel shader will need the same define.

```
#define NUM_LIGHTS 4
```

HLSL also allows the use of array with defines to determine the number of elements in the array. This array is for the four positions of the four point lights.

```
cbuffer LightPositionBuffer
{
    float4 lightPosition[NUM_LIGHTS];
};
```

The PixelInputType has been modified to hold the four updated positions of the point lights.

```
struct PixelInputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
    float3 lightPos1 : TEXCOORD1;
    float3 lightPos2 : TEXCOORD2;
    float3 lightPos3 : TEXCOORD3;
    float3 lightPos4 : TEXCOORD4;
};

PixelInputType LightVertexShader(VertexInputType input)
{
```

The position of the four lights in the world in relation to the vertex must be calculated, normalized, and then sent into the pixel shader.

```
    // Determine the light positions based on the position of the lights and the position of the vertex in the world.
    output.lightPos1.xyz = lightPosition[0].xyz - worldPosition.xyz;
    output.lightPos2.xyz = lightPosition[1].xyz - worldPosition.xyz;
    output.lightPos3.xyz = lightPosition[2].xyz - worldPosition.xyz;
    output.lightPos4.xyz = lightPosition[3].xyz - worldPosition.xyz;

    // Normalize the light position vectors.
    output.lightPos1 = normalize(output.lightPos1);
    output.lightPos2 = normalize(output.lightPos2);
    output.lightPos3 = normalize(output.lightPos3);
    output.lightPos4 = normalize(output.lightPos4);

    return output;
```

```
}
```

Light.ps

```
#define NUM_LIGHTS 4
```

This array is for the four colors of the four point lights.

```
cbuffer LightColorBuffer
{
    float4 diffuseColor[NUM_LIGHTS];
};

struct PixelInputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
    float3 normal : NORMAL;
    float3 lightPos1 : TEXCOORD1;
    float3 lightPos2 : TEXCOORD2;
    float3 lightPos3 : TEXCOORD3;
    float3 lightPos4 : TEXCOORD4;
};

float4 LightPixelShader(PixelInputType input) : SV_TARGET
{
    float4 textureColor;
    float lightIntensity1, lightIntensity2, lightIntensity3, lightIntensity4;
    float4 color, color1, color2, color3, color4;
```

The light intensity for each of the four point lights is calculated using the position of the light and the normal vector.

```
    // Calculate the different amounts of light on this pixel based on the positions of the lights.
    lightIntensity1 = saturate(dot(input.normal, input.lightPos1));
    lightIntensity2 = saturate(dot(input.normal, input.lightPos2));
    lightIntensity3 = saturate(dot(input.normal, input.lightPos3));
    lightIntensity4 = saturate(dot(input.normal, input.lightPos4));
```

The amount of color contributed by each point light is calculated from the intensity of the point light and the light color.

```
    // Determine the diffuse color amount of each of the four lights.
    color1 = diffuseColor[0] * lightIntensity1;
    color2 = diffuseColor[1] * lightIntensity2;
    color3 = diffuseColor[2] * lightIntensity3;
    color4 = diffuseColor[3] * lightIntensity4;

    // Sample the texture pixel at this location.
    textureColor = shaderTexture.Sample(SampleType, input.tex);
```

Finally add all the four point lights together to get the final light color for this pixel. Multiply the summed light color by the texture pixel and the calculations are complete.

```
    // Multiply the texture pixel by the combination of all four light colors to get the final result.
    color = saturate(color1 + color2 + color3 + color4) * textureColor;

    return color;
}
```

Lightshaderclass.h

The LightShaderClass is similar to the previous tutorials just that it has been modified to handle point lights.

The number of lights the shader uses is also defined here and has to match what is inside the shaders.

```
const int NUM_LIGHTS = 4;

class LightShaderClass
{
private:
```

There are two new structures for the diffuse color and light position arrays that are used in the vertex and pixel shader.

```
	struct LightColorBufferType
	{
		D3DXVECTOR4 diffuseColor[NUM_LIGHTS];
	};

	struct LightPositionBufferType
	{
		D3DXVECTOR4 lightPosition[NUM_LIGHTS];
	};

public:
	bool Render(ID3D11DeviceContext*, int, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX,
		ID3D11ShaderResourceView*, D3DXVECTOR4[], D3DXVECTOR4[]);

private:
	bool SetShaderParameters(ID3D11DeviceContext*, D3DXMATRIX, D3DXMATRIX, D3DXMATRIX,
		ID3D11ShaderResourceView*, D3DXVECTOR4[], D3DXVECTOR4[]);

private:
```

There are two new buffers for the diffuse color array and light position array.

```
	ID3D11Buffer* m_lightColorBuffer;
	ID3D11Buffer* m_lightPositionBuffer;
};
```

Lightshaderclass.cpp

```
LightShaderClass::LightShaderClass()
{
```

Initialize the light color and light position buffers to null in the class constructor.

```
	m_lightColorBuffer = 0;
	m_lightPositionBuffer = 0;
}
```

The render function takes in as input two new arrays for the point light diffuse color and the point light position. These two arrays will be set in the shader first before rendering takes place.

```
bool LightShaderClass::Render(ID3D11DeviceContext* deviceContext, int indexCount, D3DXMATRIX worldMatrix,
	D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView* texture,
	D3DXVECTOR4 diffuseColor[], D3DXVECTOR4 lightPosition[])
{
	// Set the shader parameters that it will use for rendering.
	result = SetShaderParameters(deviceContext, worldMatrix, viewMatrix, projectionMatrix, texture,
		diffuseColor, lightPosition);
}


bool LightShaderClass::InitializeShader(ID3D11Device* device, HWND hwnd, WCHAR* vsFilename, WCHAR*
	psFilename)
{
```

The light color array buffer is setup and created here.

```
	// Setup the description of the dynamic constant buffer that is in the pixel shader.
	lightColorBufferDesc.Usage = D3D11_USAGE_DYNAMIC;
	lightColorBufferDesc.ByteWidth = sizeof(LightColorBufferType);
	lightColorBufferDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
	lightColorBufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
	lightColorBufferDesc.MiscFlags = 0;
	lightColorBufferDesc.StructureByteStride = 0;

	// Create the constant buffer pointer so we can access the pixel shader constant buffer from within this class.
	result = device->CreateBuffer(&lightColorBufferDesc, NULL, &m_lightColorBuffer);
```

The light position array buffer is setup and created here. Note that I did make the position a four float vector so that this function won't fail to meet the multiple of 16 requirement.

```
	// Setup the description of the dynamic constant buffer that is in the vertex shader.
	lightPositionBufferDesc.Usage = D3D11_USAGE_DYNAMIC;
	lightPositionBufferDesc.ByteWidth = sizeof(LightPositionBufferType);
	lightPositionBufferDesc.BindFlags = D3D11_BIND_CONSTANT_BUFFER;
	lightPositionBufferDesc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
	lightPositionBufferDesc.MiscFlags = 0;
	lightPositionBufferDesc.StructureByteStride = 0;

	// Create the constant buffer pointer so we can access the vertex shader constant buffer from within this class.
	result = device->CreateBuffer(&lightPositionBufferDesc, NULL, &m_lightPositionBuffer);
}

bool LightShaderClass::SetShaderParameters(ID3D11DeviceContext* deviceContext, D3DXMATRIX worldMatrix,
	D3DXMATRIX viewMatrix, D3DXMATRIX projectionMatrix, ID3D11ShaderResourceView* texture,
	D3DXVECTOR4 diffuseColor[], D3DXVECTOR4 lightPosition[])
{
	LightPositionBufferType* dataPtr2;
	LightColorBufferType* dataPtr3;
	unsigned int bufferNumber;
```

The light position array buffer that will be used in the vertex shader is setup here.

```
	// Lock the light position constant buffer so it can be written to.
	result = deviceContext->Map(m_lightPositionBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0,
		&mappedResource);
	if(FAILED(result))
	{
		return false;
	}

	// Get a pointer to the data in the constant buffer.
	dataPtr2 = (LightPositionBufferType*)mappedResource.pData;

	// Copy the light position variables into the constant buffer.
	dataPtr2->lightPosition[0] = lightPosition[0];
	dataPtr2->lightPosition[1] = lightPosition[1];
	dataPtr2->lightPosition[2] = lightPosition[2];
	dataPtr2->lightPosition[3] = lightPosition[3];

	// Unlock the constant buffer.
	deviceContext->Unmap(m_lightPositionBuffer, 0);

	// Set the position of the constant buffer in the vertex shader.
	bufferNumber = 1;

	// Finally set the constant buffer in the vertex shader with the updated values.
	deviceContext->VSSetConstantBuffers(bufferNumber, 1, &m_lightPositionBuffer);

	// Set shader texture resource in the pixel shader.
	deviceContext->PSSetShaderResources(0, 1, &texture);
```

The light color array buffer that will be used in the pixel shader is setup here.

```
	// Lock the light color constant buffer so it can be written to.
	result = deviceContext->Map(m_lightColorBuffer, 0, D3D11_MAP_WRITE_DISCARD, 0,
		&mappedResource);
	if(FAILED(result))
	{
		return false;
	}

	// Get a pointer to the data in the constant buffer.
	dataPtr3 = (LightColorBufferType*)mappedResource.pData;

	// Copy the light color variables into the constant buffer.
	dataPtr3->diffuseColor[0] = diffuseColor[0];
	dataPtr3->diffuseColor[1] = diffuseColor[1];
	dataPtr3->diffuseColor[2] = diffuseColor[2];
	dataPtr3->diffuseColor[3] = diffuseColor[3];

	// Unlock the constant buffer.
	deviceContext->Unmap(m_lightColorBuffer, 0);

	// Set the position of the constant buffer in the pixel shader.
	bufferNumber = 0;

	// Finally set the constant buffer in the pixel shader with the updated values.
	deviceContext->PSSetConstantBuffers(bufferNumber, 1, &m_lightColorBuffer);
}
```

Lightclass.h

The LightClass has been changed to be a point light instead of a directional light. It has variables and helper functions for position and color which are all that are needed for point lights in this tutorial.

```
class LightClass
{
public:
	void SetPosition(float, float, float);
	D3DXVECTOR4 GetPosition();

private:
	D3DXVECTOR4 m_position;
};
```

Lightclass.cpp

```
void LightClass::SetPosition(float x, float y, float z)
{
	m_position = D3DXVECTOR4(x, y, z, 1.0f);
	return;
}

D3DXVECTOR4 LightClass::GetPosition()
{
	return m_position;
}
```

Graphicsclass.h

```
class GraphicsClass
{
private:
```

We will create four different point lights in this tutorial to implement both point lights and multiple light sources.

```
	LightClass *m_Light1, *m_Light2, *m_Light3, *m_Light4;
```

```
};
```

Graphicsclass.cpp

```
GraphicsClass::GraphicsClass()
{
```

Initialize the four point lights to null in the class constructor.

```
	m_Light1 = 0;
	m_Light2 = 0;
	m_Light3 = 0;
	m_Light4 = 0;
}

bool GraphicsClass::Initialize(int screenWidth, int screenHeight, HWND hwnd)
{
```

The model for this tutorial will be a flat plane. The idea is to put four light points in the four corners of the plane to display the effect of multiple different colored point lights lighting up a flat surface.

```
	// Initialize the model object.
	result = m_Model->Initialize(m_D3D->GetDevice(), L"./data/stone01.dds", "./data/plane01.txt");
```

The first point light will be red and placed in the upper left corner.

```
	// Create the first light object.
	m_Light1 = new LightClass;
	if(!m_Light1)
	{
		return false;
	}

	// Initialize the first light object.
	m_Light1->SetDiffuseColor(1.0f, 0.0f, 0.0f, 1.0f);
	m_Light1->SetPosition(-3.0f, 1.0f, 3.0f);
```

The second point light will be green and placed in the upper right corner.

```
	// Create the second light object.
	m_Light2 = new LightClass;
	if(!m_Light2)
	{
		return false;
	}

	// Initialize the second light object.
	m_Light2->SetDiffuseColor(0.0f, 1.0f, 0.0f, 1.0f);
	m_Light2->SetPosition(3.0f, 1.0f, 3.0f);
```

The third point light will be blue and placed in the bottom left corner.

```
	// Create the third light object.
	m_Light3 = new LightClass;
	if(!m_Light3)
	{
		return false;
	}

	// Initialize the third light object.
	m_Light3->SetDiffuseColor(0.0f, 0.0f, 1.0f, 1.0f);
	m_Light3->SetPosition(-3.0f, 1.0f, -3.0f);
```

The fourth point light will be white and placed in the bottom right corner.

```
	// Create the fourth light object.
	m_Light4 = new LightClass;
	if(!m_Light4)
	{
		return false;
	}

	// Initialize the fourth light object.
	m_Light4->SetDiffuseColor(1.0f, 1.0f, 1.0f, 1.0f);
	m_Light4->SetPosition(3.0f, 1.0f, -3.0f);
}

void GraphicsClass::Shutdown()
{
```

The four point lights are released in the Shutdown function.

```
	// Release the light objects.
	if(m_Light1)
	{
		delete m_Light1;
		m_Light1 = 0;
	}

	if(m_Light2)
	{
		delete m_Light2;
		m_Light2 = 0;
	}

	if(m_Light3)
	{
		delete m_Light3;
		m_Light3 = 0;
	}

	if(m_Light4)
	{
		delete m_Light4;
		m_Light4 = 0;
	}

}

bool GraphicsClass::Frame()
{
	// Set the position of the camera.
	m_Camera->SetPosition(0.0f, 2.0f, -12.0f);
}

bool GraphicsClass::Render()
{
	D3DXVECTOR4 diffuseColor[4];
	D3DXVECTOR4 lightPosition[4];
```

We setup the two arrays (color and position) from the four point lights. This makes it easier to just send in the two array variables instead of the eight different light components.

```
	// Create the diffuse color array from the four light colors.
	diffuseColor[0] = m_Light1->GetDiffuseColor();
	diffuseColor[1] = m_Light2->GetDiffuseColor();
	diffuseColor[2] = m_Light3->GetDiffuseColor();
	diffuseColor[3] = m_Light4->GetDiffuseColor();

	// Create the light position array from the four light positions.
	lightPosition[0] = m_Light1->GetPosition();
```

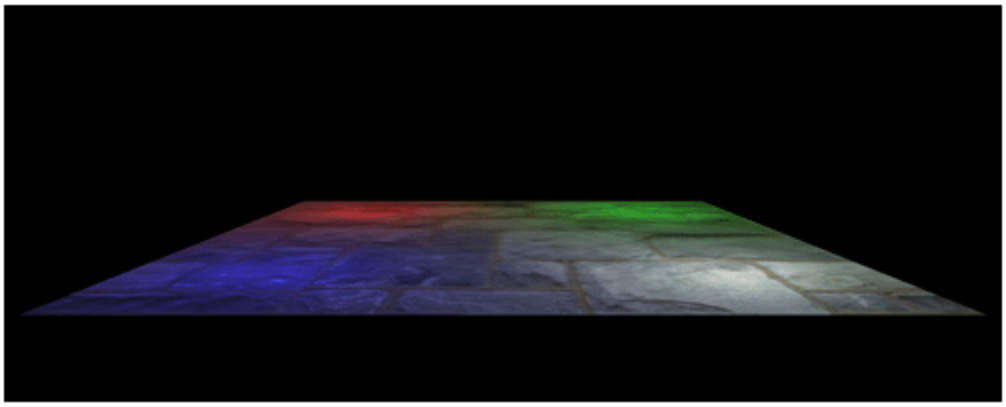
```
        lightPosition[1] = m_Light2->GetPosition();
        lightPosition[2] = m_Light3->GetPosition();
        lightPosition[3] = m_Light4->GetPosition();
```

Render the plane model using the light shader and the four point lights.

```
        // Render the model using the light shader and the light arrays.
        result = m_LightShader->Render(m_D3D->GetDeviceContext(), m_Model->GetIndexCount(), worldMatrix,
                viewMatrix, projectionMatrix, m_Model->GetTexture(), diffuseColor, lightPosition);
}
```

Summary

Now you should understand the basic concept of illuminating models with point lights and multiple light sources.



Exercises

1. Recompile and run the program. You should see a plane illuminated by four different point lights.
2. Position the red, green, and blue point lights at the same location. You should see the resulting point light is white.
3. Change the color and position of the point lights to see the different effects.
4. Add a fifth point light.
5. Only render one point light to see the effect it produces by itself.

[Original script: www.rastertek.com]